A Relational Modeling System

A. Atamtürk
E.L. Johnson
J. T. Linderoth
M.W.P. Savelsbergh

School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0205, U.S.A.

Abstract

We discuss an integer linear programming modeling system based on relational database technology. In the system, all modeling related activities, such as model formulation, model instantiation, and model and instance management, are done using the data manipulation language SQL.

September 1996
Revised March 1998

1 Introduction

Anyone who has ever attempted to apply mathematical programming in practice knows that it is usually not a simple and straightforward exercise. The road from a real-life problem situation to a satisfactory solution can be quite long and full of complications. There are many factors that contribute to this, but one of the most important is the large amount of data that needs to be handled.

Since it is standard practice in industry to store information in databases, Mitra et al. [MKLM95] argue that in order for mathematical programming to gain better acceptance as a modeling tool within corporate decision studies, a unified approach which integrates a modeling language with a relational database is necessary, as it will provide a more powerful tool for constructing models which are truly data driven. They propose to achieve this by incorporating relational database structures into the syntax of the algebraic modeling language MPL [Max93].

The goal of our research has been to develop an integrated modeling environment in which data management plays an even more central role and in which all modeling activities, such as model formulation, model instantiation, model solution, model validation, and solution analysis, are done in a common paradigm. Our efforts have resulted in a relational modeling system based on relational database technology. Model formulation, instance generation, and solution manipulation are all done using the data manipulation language SQL (Structured Query Language) [ANS, Dat87]. Many other desirable features of modeling environments, such as model and instance management and report writing are facilitated because they can often be done using available database tools. Furthermore, model builder as well as end-user can work with the same system and users can easily share models. A prototype called ARMOS (A Relational MOdeling System) has been implemented and performs well.

Several other researchers have observed the potential of relational algebra for mathematical programming programming modeling. Our research was partly motivated by the ideas presented in Johnson [Joh89]. Choobineh [Cho91] designed SQLMP. However, Choobineh extends SQL and uses the algebraic paradigm for model conceptualization, whereas we do not extend SQL and use the block-schematic paradigm for model conceptualization. The block-schematic paradigm was introduced by Baker [Bak83] and Welch [Wel87] and forms the basis of MathPro [Mat89] and MIMI [Bak92]. The reason we use the block-schematic paradigm is purely pragmatic; the block-schematic approach is easier to embed in a relational modeling scheme. We are not claiming that the block-schematic paradigm is better than the algebraic approach used in systems such as GAMS [BKM88], AMPL [FGK93], MPL [Max93], and AIMMS [BE93]. Which of the approaches to use is largely a matter of taste, although it is claimed that many industrial users, particularly in those in the process industry prefer the block-schematic paradigm as it is closer to their activity-based view of the model. Dolk [Dol88] shows how structured models (as introduced by Geoffrion [Geo87]) can be represented and manipulated easily using SQL. Dolk also discusses how SQL might be used to facilitate the solution of mathematical programming models. He expects that this will require nonstandard SQL features. Our research shows that interfacing with an optimizer can be done completely with standard SQL.

The paper is organized as follows. In Section 2, we present our view on the characteristics of good modeling environments. In Section 3, we give a brief introduction to SQL. In Section 4, we introduce the basic concepts of relational modeling. In Section 5, we show how to use these basic concepts to model the fleet assignment problem. In Section 6, we illustrate how these concepts can be implemented in a standard database environment. In Section 7, we present some conclusions and directions for future research. In the appendices, we give relational models for several well-known planning problems and an overview of ARMOS' functionality.

2 Modeling environments

A model is an abstraction of a real-life decision situation. Therefore, its solution has to be interpreted with care and not as the definitive answer to the real-life problem.

The process of generating a satisfactory solution to a real-life problem involves developing a model (which typically means making simplifying assumptions), generating an instance of the model (which typically involves gathering huge amounts of data), solving the instance (which typically involves transforming the instance data into a machine readable form), validating the solution and the model (which involves verifying the appropriateness of the simplifying assumptions), and, if the need arises, repeating these steps. In addition, models may have to be modified when changes occur in the real-life decision situation or user needs become different. This iterative process represents the *modeling life cycle* in which a model evolves over time.

A computer based linear programming modeling environment has to nurture the entire modeling life cycle, i.e., facilitate ongoing evolution of models, and has to support the management of resources used in the modeling life cycle, such as data, models, solvers, solutions.

Nowadays highly accurate data gathering and processing technologies are widely available in industry. Typically, the availability of more and more accurate data leads to the development of more detailed models, which means that data management facilities in modeling environments are crucially important. Most of the data required for an instance of a model will be stored in corporate databases and has to be processed before it can be used to construct an instance of the model at hand. Probably the most widely used and most reliable tool to handle large amounts of data is SQL.

Typically, it is necessary to solve many instances of one model with varying data. Therefore, it is important that data and model are separated, i.e., the model should be stated independently from any data. Consequently, a modeling environment should support, if not enforce, the separation of model and instance.

These are only a few, though very important, features an effective modeling environment should have. Other desirable features include support for model documentation and report writing, and the availability of different views such as lists, schemas, figures, charts of the model, the instance data, and the solution. The modeling concepts we propose are very well suited to form the basis of a modeling system that has all the desired features.

3 Structured Query Language

The data manipulation language SQL is the most popular interface to relational databases. In this section, we briefly introduce the SQL constructs used in the design and implementation of our relational modeling system.

The `CREATE TABLE` statement is used to define and create a relational data table.

```
CREATE TABLE table (
    attribute        attribute_type,
    attribute        attribute_type,
    ...
    attribute        attribute_type);
```

Examples:

```
CREATE TABLE production (
    plant            CHAR(10),
    product          CHAR(10),
    capacity         NUMBER,
    cost             NUMBER);

CREATE TABLE demand (
    center           CHAR(10),
    product          CHAR(10),
    amount           NUMBER);
```

A query to the database is formulated as follows:

```
    SELECT           attribute(s)
    FROM             table(s)
    WHERE            predicate(s)
    GROUP BY         attribute(s);
```

In SQL each query is evaluated as follows. First the cartesian product of the tables in the `FROM` clause is computed. Then the result is filtered by the predicate(s) of the `WHERE` clause. This result is then partitioned by the attribute(s) listed in the `GROUP BY` clause. The last step is the display of the results which were requested through the `SELECT` clause.

Example:

```
    SELECT plant, product, amount
    FROM production, demand
    WHERE production.product = demand.product
    GROUP BY product;
```

The last SQL construct used in the design and implementation of our relational modeling system is the *view*. A view is simply a particular look at the database. Although a view is a table, it does not exist physically in the database as a table; no storage space or data are allocated for it. The `CREATE VIEW` statement is used to define a virtual table.

Example:

```
CREATE VIEW largecap (plant, product, capacity) AS
SELECT plant, product, capacity
FROM production
WHERE capacity > 1000;
```

4 Relational modeling

We will illustrate the basic concepts of relational modeling by means of an example. We consider a production distribution problem with single sourcing requirements [MWJS78].

4.1 Problem situation

The problem is to decide how much of each product to produce at plants, how to ship to warehouses, and tranship to demand-centers subject to the constraint that a warehouse has to ship all of the demand for all products to any demand-center to which it ships. In other words, each demand-center is assigned a single warehouse that must meet all of its demand for the several products.

4.2 Instance data

The data involved in this model are production cost per product per plant, production capacity per product per plant, shipping cost from plant to warehouse, shipping cost from warehouse to demand center, and demand per product per demand center. These data are assumed to be available in a database in user data tables: `Production, Shipcost, Tranship`, and `Demand`, which are defined as follows:

```
CREATE TABLE Production (
    plant       char(10),
    product     char(10),
    capacity    number,
    cost        number);

CREATE TABLE ShipCost (
    plant       char(10),
    whse        char(10),
    cost        number);

CREATE TABLE Tranship (
    whse        char(10),
    center      char(10),
    cost        number);
```

```
CREATE TABLE Demand (
    center      char(10),
    product     char(10),
    amount      number);
```

An instance of the production-distribution problem is given by the following user data tables. This instance will be used throughout our discussion of the basic concepts of our approach.

Table `Production`:

PLANT	PRODUCT	CAPACITY	COST
topeka	chips	200	230
topeka	nachos	800	280
newyork	chips	600	255

Table `Shipcost`:

PLANT	WHSE	COST
topeka	topeka	1
topeka	newyork	45
newyork	topeka	45
newyork	newyork	2

Table `Tranship`:

WHSE	CENTER	COST
topeka	east	60
topeka	south	30
topeka	west	40
newyork	east	10
newyork	south	30
newyork	west	80

Table `Demand`:

CENTER	PRODUCT	AMOUNT
east	chips	200
east	nachos	50
south	chips	250
south	nachos	180
west	chips	150
west	nachos	300

All that has been done so far is to specify the data that is needed to define an instance of the problem. Note that nachos are only produced in Topeka, not in New York.

4.3 Column and row strips

In an integer linear program activities or decisions are modeled as variables, possibly with integrality restrictions on some of them, and restrictions and relations among the decisions are modeled as linear equations and inequalities in terms of the variables. Typically, variables in an integer linear program can be grouped into classes with similar characteristics, based on what they represent in the underlying problem situation. Similarly, the linear equations and inequalities, or constraints, can also be grouped into classes with similar characteristics. These classes of variables and classes of constraints can be used to construct a block-schematic view of the integer linear program, see for instance Welch [Wel87]. In a block-schematic view, classes of variables are called *column strips*, classes of constraints are called *row* strips, and their intersections, where interactions occur, are called *blocks*.

There are three types of decisions (classes of variables) in our production distribution model:

- How much to produce of each product at a plant?
- How much of each product to ship from a plant to a warehouse?
- Which warehouse to assign to each center?

Each type of decision will be represented by a class of variables in the model and by a column strip in the block-schematic representation. Since there are three types of decisions, there will be three column strips: `Produce, Ship`, and `Assign`.

Since each decision is related to a specific subset of the data, e.g., we have to determine how much to produce for each combination of a plant and a product, we can think of column strips as selections of data, and we can thus define them using the SQL construct of a view.

Because we ultimately have to prepare the machine readable form of the integer linear program, which requires indices, we store an index with each variable. In the underlying relational model there is no predefined order among the rows of a table, but SQL provides a pseudo-column called `RowNum`, which returns a number indicating the sequence in which a row was selected. We use the `RowNum` construct to define the unique indices for each row and column strip.

```
CREATE VIEW Produce (ix, plant, product) AS
SELECT RowNum, plant, product
FROM Production;

CREATE VIEW Ship (ix, plant, whse, product) AS
SELECT RowNum, plant, whse, product
FROM Production, Shipcost
WHERE Production.plant = Shipcost.plant;
```

```
CREATE VIEW Assign (ix, whse, center) AS
SELECT RowNum, whse, center
FROM Tranship;
```

The definition of `Ship`, for example, indicates that there will be a variable for each combination of a plant, a warehouse, and a product and that these combinations can be obtained from the user data tables `Production` and `Shipcost`. Based on the data tables of the instance specified above, the column strip `Ship` defines the following variables plus associated indices.

```
IX   PLANT      WHSE       PRODUCT
---- ---------- ---------- ----------
1    topeka     topeka     chips
2    topeka     newyork    chips
3    topeka     topeka     nachos
4    topeka     newyork    nachos
5    newyork    topeka     chips
6    newyork    newyork    chips
```

Similarly, there are three types of relations and restrictions in the model.

- Production at a plant is linked to shipping from the plant to a warehouse, i.e., everything that is produced should be shipped to some warehouse.
- Enforcement of the product flow balance, i.e., the total amount of a product shipped from plants to a warehouse should equal the total amount of a product shipped to the centers.
- Enforcement of the single sourcing requirement, i.e., each center receives all its demand from a single warehouse.

Each class of constraints will be represented by a row strip in the block-schematic representation. Since there are three classes of constraints, there will be three row strips: `Prodrow, Shiprow`, and `Centrow`.

```
CREATE VIEW Prodrow (ix, plant, product) AS
SELECT RowNum, plant, product
FROM Production;

CREATE VIEW Shiprow (ix, whse, product) AS
SELECT RowNum, whse, product
FROM Tranship, Demand
WHERE Tranship.center = Demand.center
GROUP BY whse, product;
```

```
CREATE VIEW Centrow (ix, center) AS
SELECT RowNum, center
FROM Demand;
```

Based on the data tables of the instance specified above, the row strip `ShipRow` defines the following constraints plus associated indices.

```
   IX WAREHOUSE            PRODUCT
----- -------------------- --------------------
    1 topeka               chips
    2 topeka               nachos
    3 newyork              chips
    4 newyork              nachos
```

Observe that we do not specify the number of variables in a class or the number of constraints in a class. The size of an instance is not part of the model, but determined automatically by the number of records in the user data tables.

4.4 Blocks

So far we have defined the column strips and row strips of the matrix, i.e, the classes of variables and the classes of constraints of the model. Next, we have to determine whether a class of variables interacts with a class of constraints, i.e., whether there are nonzero entries in the block defined by the associated column and row strips. This gives the blocks with the technological coefficients of the matrix.

```
CREATE VIEW Block11 (rowix, colix, coef) AS
SELECT Prodrow.ix, Produce.ix, -1
FROM Prodrow, Produce
WHERE Prodrow.product = Produce.product
AND Prodrow.plant = Produce.plant;

CREATE VIEW Block12 (rowix, colix, coef) AS
SELECT Prodrow.ix, Ship.ix, 1
FROM Prodrow, Ship
WHERE Prodrow.product = Ship.product
AND Prodrow.plant = Produce.plant;

CREATE VIEW Block22 (rowix, colix, coef) AS
SELECT Shiprow.ix, Ship.ix, -1
FROM Shiprow, Ship
WHERE Shiprow.product = Ship.product
AND Shiprow.whse = Ship.whse;
```

```
CREATE VIEW Block23 (rowix, colix, coef) AS
SELECT Shiprow.ix, Assign.ix, amount
FROM Shiprow, Assign, Demand
WHERE Shiprow.product = Demand.product
AND Shiprow.whse = Assign.whse
AND Assign.center = Demand.center;

CREATE VIEW Block33 (rowix, colix, coef) AS
SELECT Centrow.ix, Assign.ix, 1
FROM Centrow, Assign
WHERE Centrow.center = Assign.center;
```

The definition of `Block23`, for example, indicates that there will be a nonzero coefficient for each product that is shipped from a warehouse to a demand center and that the value of this coefficient is equal to the demand at this demand center, which can be found in the user data table `Demand`.

As an example, we show the intermediate table that is implicitly generated during the construction of the virtual table `Block23` just before the final selection of rowix, colix, and coef is made.

```
WHSE       PRODUCT    CENTER     COEF      COLIX     ROWIX
---------  ---------  ---------  --------  --------  --------
topeka     chips      east       200       1         1
topeka     chips      south      250       2         1
topeka     chips      west       150       3         1
topeka     nachos     east        50       1         2
topeka     nachos     south      180       2         2
topeka     nachos     west       300       3         2
newyork    chips      east       200       4         3
newyork    chips      south      250       5         3
newyork    chips      west       150       6         3
newyork    nachos     east        50       4         4
newyork    nachos     south      180       5         4
newyork    nachos     west       300       6         4
```

Observe that each block defines a set of triplets specifying the nonzero coefficients of that block, and that all triplets are specified relative to that block. Therefore, to specify the complete matrix all we have to do is impose an ordering on the column strips and row strips and add the appropriate offsets to the row and column indices appearing in the triplets.

It is convenient for us to consider information pertaining purely to a class of variables, such as objective coefficients, lower, and upper bounds, and information pertaining purely to a class of constraints, such as lower and upper bounds, as blocks as well. Since this type of information is typically referred to as belonging to the rim of the matrix, we will sometimes refer to these blocks as *rim blocks*. Note that we specify constraints using lower and upper bounds on the activity instead of using a sense and a right-hand side.

Below are the definitions of the rim blocks. Since these blocks will be part of the matrix description that will be input to an integer linear programming optimizer, we create triplets.

```
CREATE VIEW ProduceObj (rowix, colix, coef) AS
SELECT null, ix, cost
FROM Produce, Production
WHERE Produce.plant = Production.plant
AND Produce.product = Production.product;

CREATE VIEW ShipObj (rowix, colix, coef) AS
SELECT null, ix, cost
FROM Ship, Shipcost
WHERE Ship.plant = Shipcost.plant
AND Ship.whse = Shipcost.whse;

CREATE VIEW AssignObj (rowix, colix, coef) AS
SELECT null, ix, SUM(amount) * cost
FROM Assign, Demand, Tranship
WHERE Assign.center = Tranship.center
AND Assign.whse = Tranship.whse
AND Assign.center = Demand.center
GROUP BY cost;

CREATE VIEW ProduceUp (rowix, colix, coef) AS
SELECT ix, null, capacity
FROM Produce, Production
WHERE Produce.product = Production.product;

CREATE VIEW ProdrowLo (rowix, colix, coef) AS
SELECT ix, null, 0
FROM Prodrow;

CREATE VIEW ProdrowUp (rowix, colix, coef) AS
SELECT ix, null, 0
FROM Prodrow;

CREATE VIEW ShiprowLo (rowix, colix, coef) AS
SELECT ix, null, 0
FROM Shiprow;

CREATE VIEW ShiprowUp (rowix, colix, coef) AS
SELECT ix, null, 0
FROM Shiprow;
```

	RowLo	Produce	Ship	Assign	RowUp
Objective		ProduceObj	ShipObj	AssignObj	
ColumnLo		0	0	0	
ProdRow	$0 \leq$	Block11	Block12		≤ 0
ShipRow	$0 \leq$		Block22	Block23	≤ 0
CentRow	$1 \leq$			Block33	≤ 1
ColumnUp		ProduceUp	Inf	Inf	

Figure 1: Block schematic view of the production-distribution model

```
CREATE VIEW CentrowUp (rowix, colix, coef) AS
SELECT ix, null, 1
FROM Centrow;

CREATE VIEW CentrowLo (rowix, colix, coef) AS
SELECT ix, null, 1
FROM Centrow;
```

The definition of `ProduceObj` indicates that for each combination of a plant and a product defined in the column strip `Produce` the objective coefficient can be found in the `Production` user data table in the field cost of the row that matches this particular combination. The definition of `AssignObj` shows that it is also possible to have computed objective coefficients.

This completes the model description. A block-schematic view of the model is given in Figure 1.

Observe that the definition of column strips, row strips, and blocks only depends on the structure of the user data tables, *not* on the records contained in those tables. This ensures complete separation of model and data. It also means that the same model definition can handle instances with two plants, two products, two warehouses, and three demand centers, as well as instances with hundreds of plants, thousands of products, hundreds of warehouses, and millions of demand centers. Observe that an instance of a model exists as a collection of views, i.e., virtual tables. This is a major difference from systems in which instances are physically stored in a database.

When defining the column strips, row strips, and blocks of the production distribution model, we have explicitly given complete SQL statements. Using complete SQL statements does not lead to a concise description of the model. On the contrary, the description is fairly lengthy. However, a closer examination reveals that there is a lot of information contained in this description that does not pertain to the model, but is a result of the syntax of SQL. Therefore, when relational modeling concepts are embedded in a relational modeling environment, a user interface needs to be developed that shields a user from the underlying SQL syntax and reduces the effort to specify a model. We offer two suggestions for accomplishing this. First, a language and compiler, in the same spirit as AMPL and GAMS, can be developed. The language would allow concise model descriptions and the compiler contains the logic to create the complete SQL statements. Secondly, a dedicated model editor, similar to structure-based editors for programming languages, can be embedded in the environment. Such a dedicated model editor might work by providing templates in which a user enters only information relevant to the model.

4.5 Ordered domains

An important class of linear programming models involves multi-period production planning. Such models typically contain a class of balancing constraints that ensure a proper transition from one period to the next, e.g., for every period except the first, the inventory at the start of period $t-1$ plus the production in period $t-1$ minus the sales in period $t-1$ has to equal the inventory at the start of period t. Such models pose a serious problem for the relational modeling approach because it relies on a natural ordering of the data, such as weeks, months, and years.

The relational model that forms the basis of relational database implementations does not support the concept of ordered domains. There are two ways to deal with this dilemma. First, commercial implementations of a relational database have special functions related to time and we could make use of these. Second, when building a model, we can use numerical representations of the ordered domains and use SQL constructs to implement ordering concepts such as 'first', 'successor', and 'predecessor'.

As an example consider the following two user data tables. The first table is not necessary, but mainly serves as a table that can be used in the report generation phase.

Table `Date`:

```
NAME      PERIOD
--------  --------
February  2
April     4
June      6
```

Table `Production`:

```
PRODUCT   PERIOD    CAPACITY  COST
--------  --------  --------  --------
chips     2         2000      76
chips     4         1600      78
```

```
chips     6     2000    76
nachos    2     1200    82
nachos    4     1200    82
nachos    6     800     86
```

Consider the class of balancing constraints mentioned above. The column strip associated with the inventory variables can be defined as

```
CREATE VIEW Inventory (ix, product, period) AS
SELECT RowNum, product, period
FROM production;
```

We require a balance constraint for each product in each period except for the first. The row strip associated with the balancing constraints can be defined as

```
CREATE VIEW Balance (ix, product, period) AS
SELECT RowNum, product, period
FROM production
WHERE period > (SELECT min(period) FROM production);
```

We have used a subquery to determine the first period appearing in the table production. For this class of constraints, there are two interactions between the column strip and the row strip – there is product flow "into" the period and "out" of the period. To accomplish this, we simply define the two matrix blocks

```
CREATE VIEW Block_1 (rowix, colix, coef) AS
SELECT Balance.ix, Inventory.ix, -1
FROM Balance, Inventory
WHERE Balance.product = Inventory.product
AND Balance.period = Inventory.period;

CREATE VIEW Block_2 (rowix, colix, coef) AS
SELECT Balance.ix, Inventory.ix, 1
FROM Balance, Inventory
WHERE Balance.product = Inventory.product
AND Inventory.period =
    (SELECT max(period) FROM Inventory WHERE period < Balance.period);
```

Again, we have used a subquery, this time to determine the previous period appearing in the production table.

This concludes our description of the basic concepts of relational modeling. In Appendix A, we provide examples of relational models for some well-known planning problems.

5 The Fleet Assignment Problem

To provide the reader with a real-life application in which using the relational modeling scheme is a natural and convenient choice, we discuss an important planning problem faced by the airline companies. In the *fleet assignment problem*, we are given a flight schedule and a set of fleet (aircraft) types. The problem is to find a minimum cost assignment of the fleet types to the flight legs in the schedule. The flight schedule (time table) of a medium to large airline is huge and typically stored in a relational database. Below we show some of the typical information found in the database.

Table `Schedule`:

```
LEG    DEPSTA   DEPTIM   ARRSTA   ARRTIM   FLEET    COST       ...
------ -------- -------- -------- -------- -------- ------
101    DFW      745      BOS      1055     734       8270
101    DFW      745      BOS      1055     737       9198
101    DFW      745      BOS      1055     757      11088
101    DFW      745      BOS      1055     767      12098
102    BOS      1200     DFW      1500     734       8270
102    BOS      1200     DFW      1500     737       9198
102    BOS      1200     DFW      1500     757      11088
102    BOS      1200     DFW      1500     767      12098
201    DFW      745      SFO      1145     734       9653
201    DFW      745      SFO      1145     737      10731
201    DFW      745      SFO      1145     757      12936
201    DFW      745      SFO      1145     767      14036
202    SFO      1245     DFW      1615     734      10140
202    SFO      1245     DFW      1615     737      10731
202    SFO      1245     DFW      1615     757      12936
202    SFO      1245     DFW      1615     767      14036
401    DFW      1700     BOS      2000     737       9198
401    DFW      1700     BOS      2000     757      11088
401    DFW      1700     BOS      2000     767      12098
402    BOS      2130     SFO      230      737      17520
402    BOS      2130     SFO      230      757      21120
402    BOS      2130     SFO      230      767      23830
403    SFO      245      DFW      615      737      10731
403    SFO      245      DFW      615      757      12936
403    SFO      245      DFW      615      767      14036
...
```

In this particular example, we see that there is flight from Dallas/Fort Worth International Airport to Logan Airport in Boston departing at 7.45am and arriving at 10.55am and that this flight can be flown by either a Boeing 734 at a cost of 8270, a Boeing 737 at a cost of 9198, a Boeing 757 at a cost of 11088, and a Boeing 767 at a cost of 12098.

The information on available fleet types is also stored in a relational database and looks somewhat like this

Table `Fleet`:

```
FLEET   AVAIL     ...
------  ------
734     6         ...
737     6         ...
757     2         ...
767     9         ...
...
```

A feasible fleet assignment must assign a fleet type to each flight leg, cannot use more aircrafts of a given fleet type than are available, and must ensure that aircrafts that arrive at a station either depart or stay on the ground and similarly that aircrafts depart from a station where they landed earlier.

To model the fleet assignment problem, we introduce two classes of variables. First, a class of binary variables indicating for each combination of flight leg and fleet type whether or not the fleet type is assigned to the flight leg. Second, a class of integer variables counting the number of aircrafts of a specific fleet type on the ground at a particular station and time.

```
CREATE VIEW Assign (ix, leg, fleet) AS
SELECT RowNum, leg, fleet
FROM   Schedule;

CREATE VIEW GroundArc (ix, fleet, station, time) AS
SELECT RowNum, Fleets.fleet, arrsta, arrtime
FROM   Fleets, Schedule
UNION
SELECT RowNum, Fleets.fleet, depsta, deptime
FROM   Fleets, Schedule;
```

Note that we create a so-called *ground arc* for each fleet type at each station and each event (arrival or departure) at that station. Ground arcs, as the name suggests are used to model aircrafts that stay on the ground at a station.

There are three classes of constraints. First, which is at the heart of the problem, we have ensure that each flight leg is assigned exactly one fleet type.

```
CREATE VIEW Cover (ix, leg) AS
SELECT RowNum, leg
FROM   Schedule
GROUP BY leg;
```

Second, we have to ensure that the fleet assignments are consistent, in the sense that there is flow balance for each fleet type at each station and each event occuring at that station. That is to say that for each fleet type, the number of aircrafts of that fleet type "arriving" at a station (either via an incoming flight leg or via a ground arc) is equal to the number of aircrafts "departing" from the station (either via an outgoing flight leg or a ground arc).

```
CREATE VIEW Balance (ix, fleet, station, time) AS
SELECT ALL
FROM   GroundArc;
```

Finally, we have to ensure for each fleet type that we do not use more aircrafts than there are available.

```
CREATE VIEW AircraftCount (ix, fleet) AS
SELECT RowNum, fleet
FROM   Fleets;
```

Now we define the blocks of the matrix. Since we have to assign exactly one fleet type to each flight leg we have the following block

```
CREATE VIEW CoverBlock (rowix, colix, coef) AS
SELECT Cover.ix, Assign.ix, 1
FROM   Cover, Assign
WHERE  Cover.leg = Assign.leg;
```

Next, we consider the blocks that define the flow balance constraints for each fleet type at each event occuring at a station. This involves selecting appropriate incoming and outgoing arcs.

First, we create a block handling the outgoing flight legs.

```
CREATE VIEW BalanceAssignDep (rowix, colix, coef) AS
SELECT Balance.ix, Assign.ix, 1
FROM   Balance, Assign
WHERE  Assign.leg = SELECT leg FROM Schedule
                    WHERE Balance.fleet = Schedule.fleet
                    AND   Balance.station = Schedule.depsta
                    AND   Balance.time = Schedule.deptime;
```

Next, we create a block handling the incoming flight legs.

```
CREATE VIEW BalanceAssignArr (rowix, colix, coef) AS
SELECT Balance.ix, Assign.ix, -1
FROM   Balance, Assign
WHERE  Assign.leg = SELECT leg FROM Schedule
                    WHERE Balance.fleet = Schedule.fleet
                    AND   Balance.station = Schedule.arrsta
                    AND   Balance.time = Schedule.arrtime;
```

Now, we switch to ground arcs and start with outgoing ground arcs.

```
CREATE VIEW BalanceGroundOut (rowix, colix, coef) AS
SELECT Balance.ix, Ground.ix, 1
FROM   Balance, Ground
WHERE  Balance.fleet = Ground.fleet
AND    Balance.station = Ground.station
AND    Balance.time = Ground.time;
```

We follow with the incoming ground arcs. Selecting the incoming arcs is a bit more involved, since it requires the identification of the previous event at a station, which is done by means of a subquery. Furthermore, we need to distinguish the first event at a station from the other events, since the previous event of the first event actually occurs as the last event occuring (of the previous day).

```
CREATE VIEW BalanceGroundIn (rowix, colix, coef) AS
SELECT Balance.ix, Ground.ix, -1
FROM   Balance, Ground
WHERE  Balance.fleet = Ground.fleet
AND    Balance.station = Ground.station
AND    Ground.time =
       SELECT max(Ground.time) FROM Ground
       WHERE  Balance.fleet = Ground.fleet
       AND    Balance.station = Ground.station
       AND    Ground.time < Balance.time;


CREATE VIEW BalanceGroundInFirst (rowix, colix, coef) AS
SELECT Balance.ix, Ground.ix, -1
FROM   Balance, Ground
WHERE  Balance.fleet = Ground.fleet
AND    Balance.station = Ground.station
AND   (Balance.time =
       SELECT min(Balance.time) FROM Balance
       WHERE  Balance.fleet = Ground.fleet
       AND    Balance.station = Ground.station)
AND    Ground.time =
       SELECT max(Ground.time) FROM Ground
       WHERE  Balance.fleet = Ground.fleet
       AND    Balance.station = Ground.station;
```

To ensure for each fleet type that we do not use more aircrafts than there are available, we take a snapshot at midnight and count all the aircrafts of a specific fleet type. Since the balancing constraints ensure that the flow is a circulation the number of aircrafts in use will be the same throughout the day and taking a snapshot at midnight suffices. There are two blocks: one to account for the 'red eye' flights, i.e., flight legs corresponding to flights that are in the air at midnight, and one for the ground arcs that cross midnight.

```
CREATE VIEW RedEyeCount (rowix, colix, coef) AS
SELECT PCount.ix, Assign.ix, 1
FROM   PCount, Assign
WHERE  PCount.fleet = Assign.fleet
AND    Assign.leg = SELECT leg FROM Schedule
                    WHERE  Schedule.arrtime < Schedule.deptime;

CREATE VIEW GroundCount (rowix, colix, coef) AS
SELECT PCount.ix, Ground.ix, 1
FROM   PCount, Ground
WHERE  PCount.fleet = Ground.fleet
AND    (Ground.station, Ground.time) IN
       SELECT station, MAX(time) FROM Ground
       GROUP BY station;
```

Specifying the rim blocks, i.e., objective function coefficients and lower and upper bounds on variables and constraints is even easier.

The above example demonstrates the ease with which it possible, in this particular case, to set up the fleet assignment model. Note also that nothing needs to be done to connect the model to the data and that no special data files need to be prepared. This is a major advantage of the proposed scheme over other modeling approaches, where it is always necessary to either prepare the data in a specific format or to somehow specify how the model can connect and extract the data from an external database. Furthermore, whenever the schedule or fleet databases are updated during the year the model can be resolved automatically. Nothing needs to be changed or done.

6 A Relational Modeling System

The preceding sections have shown the conceptual viability of modeling linear and integer programs using a relational scheme. In this section, we describe the design of a modeling system that supports the relational modeling paradigm.

6.1 Model Management

In the block schematic approach, a mathematical programming model is specified entirely in terms of row strips, column strips, and matrix blocks. In a relational database environment we can conveniently manage this information for many models. We create four “system” tables: `SysModels`, `SysRows`, `SysCols`, and `SysBlocks` that contain all the information about the views defining the various models.

The `SysModels` table contains the names of the models present in the system. It has attributes `Model` and `ObjSense`. The `Model` attribute is the unique name of a model and the `ObjSense` attribute is MAX or MIN indicating whether the specified model is a maximization or minimization problem.

The `SysRows` table contains the names of the row strips present in the system. It has attributes `Model` and `RowStrip`. The `Model` attribute is the unique name of a model and the `RowStrip` attribute is the unique name of a view defining a row strip of the specified model.

The `SysCols` table contains the names of the column strips present in the system. It has attributes `Model`, `RowStrip`, and `Type`. The `Model` attribute is the unique name of a model, the `ColStrip` attribute is the unique name of a view defining a column strip of the specified model, and the `Type` attribute is CONTINUOUS, BINARY, or INTEGER indicating the variable type associated with the specified column strip.

The `SysBlocks` table contains the names of the blocks present in the system. It has attributes `Model`, `Block`, `RowStrip`, `ColStrip`, and `Type`. The `Model` attribute is the unique name of a model, the `Block` attribute is the unique name of a matrix block of the specified model, the `RowStrip` attribute is the unique name of the view defining the row strip associated with the specified block, the `ColStrip` attribute is the unique name of the view defining the column strip associated with the specified block, and the `Type` attribute is ROWLOWER, ROWUPPER, COLOBJ, COLLOWER, COLUPPER, or BLOCKDATA indicating the type of the specified block.

The tables below show the relevant entries in the system tables pertaining to the production distribution model presented in the preceding sections.

Table `SysModels`:

```
Model      ObjSense
---------- ----------
Prod-Dist  MIN
...        ...
```

Table `SysRows`:

```
Model      RowStrip
---------- ----------
Prod-Dist  Prodrow
Prod-Dist  Shiprow
Prod-Dist  Centrow
...        ...
```

Table `SysCols`:

```
Model      RowStrip   Type
---------- ---------- ----------
Prod-Dist  Produce    CONTINUOUS
Prod-Dist  Ship       CONTINUOUS
Prod-Dist  Assign     BINARY
...        ...        ...
```

Table `SysBlocks`:

Model	Block	RowStrip	ColStrip	Type
Prod-Dist	Block11	Prodrow	Produce	BLOCKDATA
Prod-Dist	Block12	Prodrow	Ship	BLOCKDATA
Prod-Dist	Block22	Shiprow	Ship	BLOCKDATA
Prod-Dist	Block23	Shiprow	Assign	BLOCKDATA
Prod-Dist	Block33	Centrow	Assign	BLOCKDATA
Prod-Dist	ProduceObj		Produce	COLOBJ
Prod-Dist	ShipObj		Ship	COLOBJ
Prod-Dist	AssignObj		Assign	COLOBJ
Prod-Dist	ProduceUp		Produce	COLUPPER
Prod-Dist	ProdrowUp	Prodrow		ROWUPPER
Prod-Dist	ProdrowLo	Prodrow		ROWLOWER
Prod-Dist	ShiprowUp	Shiprow		ROWUPPER
Prod-Dist	ShiprowLo	Shiprow		ROWLOWER
Prod-Dist	CentrowUp	Centrow		ROWUPPER
Prod-Dist	CentrowLo	Centrow		ROWLOWER
...	...	...	...	...

6.2 Instance Management

In the production distribution problem used to illustrate the relational modeling scheme, we have used specific data tables in the definition of the model, e.g., Production, Ship-Cost, Tranship, and Demand, even though we only used the structure of these tables. It is good practice, however, to define models completely independent of its instances. To do so, we make use of another feature of SQL called a *synonym*. A synonym is an alias assigned to a table or view that may thereafter be used to refer to it. For each data table required in the definition of a model, we introduce a synonym and all references to data tables are made through these synonyms. Then, to create a specific instance of a model, all that needs to be done is to update the synonyms so that they refer to the actual data tables defining the instance.

The relational database environment is well suited to manage many instances of the same model. We create two system tables: `SysDataTables` and `SysInstances`.

The `SysDataTables` table contains the names of the active user data tables for a model. It has attributes `Model`, `BaseTable`, `Syn`, and `ActiveTable`. The `Model` attribute is the unique name of a model, the `Block` attribute is the unique name of a matrix block of the The `Model` attribute is the unique name of a model, the `BaseTable` attribute is the unique name of a special data table, called base table, having the same structure as an instance data table required in the definition of the specified model, the `Syn` attribute is the synonym for the base table used in the model definition, and the `ActiveTable` attribute is the name of the current data table associated with the specified synonym.

As mentioned before, the definition of a model depends only on the structure of the user data tables, not on the records contained in those tables. Therefore, to completely separate model and data, we use artificial tables in the model definition. The artificial tables, which we call base tables, have the same structure as the user data tables, but will always be empty. The use of base tables also gives the system a level of error-checking. When we attempt to associate a synonym with a user specified instance data table, we can check if this table has the proper structure by comparing it to the base table associated with the synonym.

Now back to our production distribution example. Instead of using the tables `Production`, `ShipCost`, `Tranship`, and `Demand` directly in the definition of the model, we create (empty) base tables `base_Production`, `base_ShipCost`, `base_Tranship`, and `base_Demand` (with the same structure) and synonyms `syn_Production`, `syn_ShipCost`, `syn_Tranship`, and `syn_Demand` (initially pointing to the base tables), and use the synonyms in the definition of the model. Then to instantiate the model, we let the synonyms point to the real data tables.

Table `SysDataTables`:

```
Model      BaseTable       Syn            CurData
---------- --------------- -------------- ----------
Prod-Dist  base_Production syn_Production Production
Prod-Dist  base_ShipCost   syn_ShipCost   ShipCost
Prod-Dist  base_Tranship   syn_Tranship   Tranship
Prod-Dist  base_Demand     syn_Demand     Demand
```

The `SysInstances` table contains the names of the (user data) tables of an instance. It has attributes `Model`, `Instance`, `Syn`, and `DataTable`. The `Model` attribute is the unique name of a model, the `Instance` attribute is a unique name of an instance of the specified model, the `Syn` attribute is the name of a synonym used in the definition of the specified model, and the `DataTable` attribute is the name of the data table associated with the specified synonym in the specified instance.

Table `SysInstances`:

```
Model      Instance        Syn            CurData
---------- --------------- -------------- -------------
Prod-Dist  PD_January      syn_Production Jan_Production
Prod-Dist  PD_January      syn_ShipCost   ShipCost
Prod-Dist  PD_January      syn_Tranship   Tranship
Prod-Dist  PD_January      syn_Demand     Jan_Demand
Prod-Dist  PD_February     syn_Production Feb_Production
Prod-Dist  PD_February     syn_ShipCost   ShipCost
Prod-Dist  PD_February     syn_Tranship   Tranship
Prod-Dist  PD_February     syn_Demand     Feb_Demand
...        ...             ...            ...
```

Finally, there are two system tables that contain solution information: `SysRuns` and `SysSols`.

The `SysRuns` table has attributes `Model`, `Instance`, `Solver`, `RunDate`, `RunTime`, `Obj`, and `CpuTime`. The `Model` attribute is the unique name of a model, the `Instance` attribute is a unique name of an instance of the specified model, the`Solver` attribute indicates the solver used for the run, the `RunDate` attribute is the system date of a particular run of the specified model for the specified instance, the `RunTime` attribute is the system time of the run in format 'hh:mm:ss', the `Obj` attribute is the objective function value obtained in the run, the `CpuTime` attribute is the CPU for the run.

Table `SysRuns`:

```
MODEL      INSTANCE   SOLVER RUNDATE RUNTIME  OBJ     CPUTIME
---------- ---------- ------ ------- -------- ------- --------
Prod-Dist  PD_January OSL    1/1/98  13:14:15 324130  00:02:14
...        ...        ...    ...     ...      ...     ...
```

The `SysSols` table has attributes `Model`, `Instance`, `RunDate`, `RunTime`, `StripName`, and `TableName`. The `Model` The `Model` attribute is the unique name of a model, the `Instance` attribute is a unique name of an instance of the specified model, the `RunDate` attribute is the system date of a particular run of the specified model for the specified instance, the `RunTime` attribute is the system time of the run in format 'hh:mm:ss', the `StripName` attribute is a row or a column strip name of the specified model, and the `TableName` attribute is the name of the data table containing the solution information for the specified row or column strip obtained in the run.

Table `SysSols`:

```
MODEL     INSTANCE    RUNDATE  RUNTIME  STRIPNAME TABLENAME
--------- ----------- -------- -------- --------- ---------
Prod-Dist PD_January  1/1/98   13:14:15 Ship      sol_Ship
...       ...         ...      ...      ...       ...
```

Table `sol_Ship`:

```
PLANT      WHSE       PRODUCT    VALUE
---------- ---------- ---------- ----------
topeka     topeka     chips        200
topeka     newyork    chips          0
topeka     topeka     nachos       480
topeka     newyork    nachos        50
newyork    topeka     chips        200
newyork    newyork    chips        200
```

Observe that the solution is put in a collection of tables. It is also possible, and in fact very easy, to put the solution immediately into the appropriate user data tables. We have chosen for the above design because it is more flexible and puts control in the hands of the user.

When the values of the decision variables have been returned to the user data tables, SQL provides a convenient tool for viewing the results of the optimization. In particular, one can easily scan subsets of the solution which may be of interest. For example, the production facility manager in Topeka can easily determine his production requirements and the total production cost by the following two queries (where we assume that the levels of production determined by the optimizer, i.e., the values of `Produce`, have been put in an additional field `amount` in the data table `Production`).

```
SELECT product, amount
FROM production
WHERE plant = 'topeka';

SELECT SUM(cost*amount)
FROM production
WHERE plant = 'topeka';
```

6.3 Solver management

Another feature of the relational modeling environment that is easily incorporated is to allow users to vary solver parameters, by defining a table `SysParams` to hold these parameters.

The `SysParams` table has attributes `Solver`, `Parameter`, and `Value`. The `Solver` attribute is the name of a solver, the `Parameter` attribute is the name of a parameter that can be set for the specified solver, and `Value` is the current value of the parameter.

Table `SysParams`:

Solver	Parameter	Value
CPLEX	CPX_PARAM_CLIQUES	1
CPLEX	CPX_PARAM_NODELIM	1000000
OSL	rtolpinf	0.00001
...	...	...

The system tables introduced above form the basis of the prototype relational modeling system ARMOS described in the next section. It should be noted that the system tables are created only once at the installation of the system and then used by the system as an internal database of existing models, instances, and solutions. Maintaining the system tables is a responsibility of the system, not of the user of the system.

6.4 ARMOS

We have developed a small prototype system called **ARMOS** that implements the ideas described in the previous sections. **ARMOS** offers a simple user interface that is coded in Embedded SQL [Ora92] and that allows a user to list the models stored in system, to load a model, to list the instances stored in the system for the loaded model, to make an instance active, to optimize the active instance, and to display solution values. The user can also display the model's matrix block structure, and view the coefficients of any particular matrix block of an active instance. Currently, both linear and mixed integer linear programs can be solved. **ARMOS** is built on top of the commercial software package OSL [DSV]. A brief description of its functionality can be found in Appendix B; a detailed description of the **ARMOS** user interface can be found in the manual *Using ARMOS, A Relational MOdeling System* [AJLS96].

As mentioned above, **ARMOS** is a prototype. It provides only the most basic functions and it only has a simple text-based user interface. There is no dedicated graphical editor supporting model development. Our goal in developing **ARMOS** was to verify the viability of using the relational modeling scheme in an actual implementation.

7 Discussion

This paper is primarily a 'proof of concept' demonstration – it attempts to show that it is possible to develop a modeling environment for mathematical programming using a single paradigm: relational database technology.

We feel there are several advantages to such an approach. It is often observed, see for example Hürliman [Hür91], that despite recent developments mathematical programming is still not fully exploited in practice. By using the widely available and well-known data manipulation language SQL for model definition as well as data definition, modelers do not need to learn a new language and can keep on using many of the available SQL tools for report writing and what-if type analysis. Furthermore, it is easy to set things up in such a way that using a model has a 'fill-in-the-blank' feel, where solution values are immediately imported into the appropriate data tables. This will be very appealing to end-users, and will increase their acceptance level. (We have all witnessed the acceptance of spread-sheet like interfaces!) It has also been observed, see for example Mitra et al. [MKLM95], that the data in corporate information systems are often regularly revised and that it is therefore desirable that a decision making system should automatically update an instance when such changes are made. In a relational modeling system, where models and instances reside in the same corporate data base, this feature is naturally available.

Finally, a few words on how the proposed relational modeling system compares to other systems based on the block-schematic model building paradigm, such as MIMI

[Bak92] MathPro and [Mat89]. Obviously, both MIMI and MathPro are, at the moment, far easier to use than our prototype system due to their more sophisticated graphical user interfaces. MIMI also has a much larger functionality, since it includes an expert system component for rule-based model solution. On the other hand, MIMI supports only two-dimensional tables and therefore requires a hierarchy of table to represent high dimensional strips or blocks. ARMOS does impose this restriction, since SQL easily handles tables with multiple fields. Furthermore, MIMI requires all relevant data to be in its own private internal database, which requires copying/transfering data from the corporate database to MIMI's database. ARMOS is installed on top of the corporate database it eliminates the data transfers and storage duplication.

We see, as the main advantage of our approach, the fact that the careful design allows for easy implementation, easy maintainability, and easy extendability. For example, the code to generate and pass an instance of a model to a solver is only a couple of lines of embedded SQL code. Model management, instance management, and solver management were very easy to add, when the core system was in place.

References

[AJLS96] Alper Atamturk, Ellis Johnson, Jeff Linderoth, and Martin Savelsbergh. Using ARMOS, a Relational MOdeling System. 1996.

[ANS] ANSI. *Database Language SQL, Document ANSI X3.135-1986.* Also avaliable as ISO document ISO/TC97/SC21/WG3 N117.

[Bak83] T. E. Baker. RESULT: An interactive modeling systems for planning and scheduling, 1983. Presented at the ORSA/TIMS meeting, Chicago, IL.

[Bak92] T. Baker. *MIMI/LP User Manual.* Chesapeake Decision Science, Inc., 1992.

[BE93] J. Bisschop and R. Entriken. *AIMMS The Modeling System.* Paragon Decision Technology, 1993.

[BKM88] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS, A User's Guide.* The Scientific Press, Redwood City, CA, 1988.

[Cho91] Joobin Choobineh. SQLMP: A data sublanguage for representation and formulation of linear mathematical models. *ORSA Journal on Computing*, 3(4):358–375, 1991.

[Dat87] C. Date. *A Guide to the SQL Standard.* Addison/Wesley, Reading, MA, 1987.

[Dol88] Daniel R. Dolk. Model management and structured modeling: The role of an information resource dictionary system. *Communications of the ACM*, 31(6):704–718, 1988.

[DSV] Julie Druckerman, David Silverman, and Kathy Viaropulos. *Optimization Subroutine Library Release 2 Guide and Reference*. IBM.

[FGK93] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL. A Modeling Language for Mathematical Programming*. The Scientific Press, 1993.

[Geo87] A. M. Geoffrion. An introduction to structured modeling. *Management Science*, 33(5):547–588, 1987.

[GM92] Harvey J. Greenburg and Frederic H. Murphy. A comparison of mathematical programming modeling systems. *Annals of Operations Research*, 38:177–238, 1992.

[Hür91] T. Hürlimann. Linear modeling tools. Working Paper 187, Institute for Automation and Operations Research, University of Fribourg, Switzerland, 1991.

[Joh89] Ellis L. Johnson. Modeling and strong linear programs for mixed integer programming. In S. W. Wallace, editor, *Algorithms and Model Formulations in Mathematical Programming*, pages 1–43. Springer-Verlag, Berlin, 1989. NATO ASI Series, Vol. F51.

[Mat89] MathPro, Inc. *MathPro Usage Guide: Introdcution and reference*, 1989.

[Max93] Maximal Software. *MPL Modeling System*, 1993.

[MKLM95] G. Mitra, B. Kristjansson, C. Lucas, and S. Moody. Sets and indices in linear programming modelling and their integration with relational data models. *Computational Optimization and Applications*, 4:263–292, 1995.

[MWJS78] T. G. Mairs, G. W. Wakefield, E. L. Johnson, and K. Speilbergh. On a production allocation and distribution problem. *Management Science*, 24:1622–1630, 1978.

[Ora92] Oracle Corporation. *Programmer's Guide to the ORACLE Precompilers*, December 1992. Version 1.5.

[Wag75] H. Wagner. *Principles of Operations Research*. Prentice-Hall, Englewood Cliffs, NJ, 1975.

[Wel87] J. S. Welch. PAM – A practicioners' approach to modeling. *Management Science*, 33(5):610–625, 1987.

Appendix A

In this appendix, we give relational models for some well-known planning problems.

The Production and Distribution Model

Here we show how the SQL commands necessary to load the production and distribution model used as working example in the preceding sections into ARMOS. The data tables describing an instance of the model are just as described in the paper.

```
create table base_pd_production(
plant char(10),
product char(10),
capacity number,
cost number
);

create table base_pd_shipcost(
plant char(10),
whse char(10),
cost number
);

create table base_pd_tranship(
whse char(10),
center char(10),
cost number
);

create table base_pd_demand(
center char(10),
product char(10),
amount number
);

create synonym syn_pd_production for base_pd_production;
create synonym syn_pd_shipcost for base_pd_shipcost;
create synonym syn_pd_tranship for base_pd_tranship;
create synonym syn_pd_demand for base_pd_demand;

insert into sysmodels values ('pd', 'min');

insert into sysdata values ('pd', 'base_pd_production', 'syn_pd_production', null);
insert into sysdata values ('pd', 'base_pd_shipcost', 'syn_pd_shipcost', null);
insert into sysdata values ('pd', 'base_pd_tranship', 'syn_pd_tranship', null);
insert into sysdata values ('pd', 'base_pd_demand', 'syn_pd_demand', null);

insert into sysrows values('pd', 'pd_prodrow');
```

```
insert into sysrows values('pd', 'pd_shiprow');
insert into sysrows values('pd', 'pd_centrow');

insert into syscols values('pd', 'pd_produce', 'continuous');
insert into syscols values('pd', 'pd_ship', 'continuous');
insert into syscols values('pd', 'pd_assign', 'continuous');

insert into sysblocks values('pd', 'pd_produce_ub', null, 'pd_produce', null, 'colupper');
insert into sysblocks values('pd', 'pd_prodrow_lb', 'pd_prodrow', null, null, 'rowlower');
insert into sysblocks values('pd', 'pd_prodrow_ub', 'pd_prodrow', null, null, 'rowupper');
insert into sysblocks values('pd', 'pd_shiprow_lb', 'pd_shiprow', null, null, 'rowlower');
insert into sysblocks values('pd', 'pd_shiprow_ub', 'pd_shiprow', null, null, 'rowupper');
insert into sysblocks values('pd', 'pd_centrow_lb', 'pd_centrow', null, null, 'rowlower');
insert into sysblocks values('pd', 'pd_centrow_ub', 'pd_centrow', null, null, 'rowupper');
insert into sysblocks values('pd', 'pd_produce_obj', null, 'pd_produce', null, 'colobj');
insert into sysblocks values('pd', 'pd_ship_obj', null, 'pd_ship', null, 'colobj');
insert into sysblocks values('pd', 'pd_assign_obj', null, 'pd_assign', null, 'colobj');

insert into sysblocks values('pd', 'pd_block11', 'pd_prodrow', 'pd_produce', null,
                             'blockdata');
insert into sysblocks values('pd', 'pd_block12', 'pd_prodrow', 'pd_ship', null, 'blockdata');
insert into sysblocks values('pd', 'pd_block22', 'pd_shiprow', 'pd_ship', null, 'blockdata');
insert into sysblocks values('pd', 'pd_block23', 'pd_shiprow', 'pd_assign', 'pd_demand',
                             'blockdata');
insert into sysblocks values('pd', 'pd_block33', 'pd_centrow', 'pd_assign', null, 'blockdata');

rem ############
rem COLUMN VIEWS
rem ############

create view pd_produce (ix, plant, product) as
select rownum, plant, product
from syn_pd_production;

create view pd_ship (ix, plant, whse, product) as
select rownum, syn_pd_production.plant, whse, product
from syn_pd_production, syn_pd_shipcost
where syn_pd_production.plant = syn_pd_shipcost.plant;

create view pd_assign (ix, whse, center) as
select rownum, whse, center
from syn_pd_tranship;

rem #########
rem ROW VIEWS
rem #########

create view pd_prodrow (ix, plant, product) as
select rownum, plant, product
```

```
from syn_pd_production;

create view tmp_pd_shiprow (whse, product) as
select whse, product
from syn_pd_tranship, syn_pd_demand
group by whse, product;

create view pd_shiprow (ix, whse, product) as
select rownum, whse, product
from tmp_pd_shiprow;

create view tmp_pd_centrow(center) as
select center from syn_pd_demand
group by center;

create view pd_centrow (ix, center) as
select rownum, center
from tmp_pd_centrow;

rem #########
rem OBJECTIVE
rem #########

create view pd_produce_obj (rowix, colix, coef) as
select null, ix, cost
from pd_produce, syn_pd_production
where pd_produce.plant = syn_pd_production.plant
and pd_produce.product = syn_pd_production.product;

create view pd_ship_obj (rowix, colix, coef) as
select null, ix, cost
from pd_ship, syn_pd_shipcost
where pd_ship.plant = syn_pd_shipcost.plant
and pd_ship.whse = syn_pd_shipcost.whse;

create view pd_assign_obj (rowix, colix, coef) as
select null, ix, sum(amount) * cost
from pd_assign, syn_pd_demand, syn_pd_tranship
where pd_assign.center = syn_pd_tranship.center
and pd_assign.whse = syn_pd_tranship.whse
and syn_pd_demand.center = pd_assign.center
group by ix, cost;

rem ######
rem BOUNDS
rem ######

create view pd_produce_ub (rowix, colix, coef) as
select null, ix, capacity
```

```
from pd_produce, syn_pd_production
where pd_produce.product = syn_pd_production.product;

create view pd_prodrow_lb (rowix, colix, coef) as
select ix, null, 0
from pd_prodrow;

create view pd_prodrow_ub (rowix, colix, coef) as
select ix, null, 0
from pd_prodrow;

create view pd_shiprow_lb (rowix, colix, coef) as
select ix, null, 0
from pd_shiprow;

create view pd_shiprow_ub (rowix, colix, coef) as
select ix, null, 0
from pd_shiprow;

create view pd_centrow_ub (rowix, colix, coef) as
select pd_centrow.ix, null, 1
from pd_centrow;

create view pd_centrow_lb (rowix, colix, coef) as
select pd_centrow.ix, null, 1
from pd_centrow;

rem ##################
rem MATRIX BLOCK VIEWS
rem ##################

create view pd_block11 (rowix, colix, coef) as
select pd_prodrow.ix, pd_produce.ix, -1
from pd_prodrow, pd_produce
where pd_prodrow.product = pd_produce.product
and pd_prodrow.plant = pd_produce.plant;

create view pd_block12 (rowix, colix, coef) as
select pd_prodrow.ix, pd_ship.ix, 1
from pd_prodrow, pd_ship
where pd_prodrow.product = pd_ship.product
and pd_prodrow.plant = pd_ship.plant;

create view pd_block22 (rowix, colix, coef) as
select pd_shiprow.ix, pd_ship.ix, -1
from pd_shiprow, pd_ship
where pd_shiprow.product = pd_ship.product
and pd_shiprow.whse = pd_ship.whse;
```

```
create view pd_block23 (rowix, colix, coef) as
select pd_shiprow.ix, pd_assign.ix, amount
from pd_shiprow, pd_assign, syn_pd_demand
where pd_shiprow.product = syn_pd_demand.product
and pd_shiprow.whse = pd_assign.whse
and pd_assign.center = syn_pd_demand.center;

create view pd_block33 (rowix, colix, coef) as
select pd_centrow.ix, pd_assign.ix, 1
from pd_centrow, pd_assign
where pd_centrow.center = pd_assign.center;
```

The model definition above illustrates the use of temporary views. A temporary view has been used to create the rowstrip `pd_shiprow`. We want a row in our constraint matrix for every *unique* combination of a warehouse and a product. Due to the form of our instance data tables `tranship` and `demand`, we need the SQL construct *group by* to accomplish this results.

```
select whse, product
from tranship, demand
group by whse, product;

WHSE                 PRODUCT
-------------------- --------------------
new york             chips
new york             nachos
topeka               chips
topeka               nachos
```

However, the `Rownum` construct necessary to create unique indices within a row or column strip does not work with the `group by` clause. Therefore, we create a temporary view using the select statement above and them query the temporary view with the `Rownum` construct to assign the unique indices to the rowstrip.

The Diet Problem

This example is the simple and famous diet problem from linear programming. The problem is to choose certain amounts of foods to eat to minimize the total food cost, while still meeting nutritional requirements. For further explanation, an excellent description of this problem is in [FGK93].

```
create table base_diet_nutr(
vitamin char(20),
n_min number,
n_max number
```

```
);

create table base_diet_food(
food char(20),
f_min number,
f_max number,
cost number
);

create table base_diet_amount(
food char(20),
vitamin char(20),
amount number
);

create synonym syn_diet_nutr for base_diet_nutr;
create synonym syn_diet_food for base_diet_food;
create synonym syn_diet_amount for base_diet_amount;

insert into sysmodels values ('diet', 'min');

insert into sysdata values ('diet', 'base_diet_nutr', 'syn_diet_nutr', null);
insert into sysdata values ('diet', 'base_diet_food', 'syn_diet_food', null);
insert into sysdata values ('diet', 'base_diet_amount', 'syn_diet_amount', null);

insert into sysrows values ('diet', 'diet_req');
insert into syscols values ('diet', 'diet_buy', 'continuous');

insert into sysblocks values ('diet', 'diet_buy_obj', null, 'diet_buy', 'food', 'colobj');
insert into sysblocks values ('diet', 'diet_buy_lb', null, 'diet_buy', 'food', 'collower');
insert into sysblocks values ('diet', 'diet_buy_ub', null, 'diet_buy', 'food', 'colupper');
insert into sysblocks values ('diet', 'diet_req_lb', 'diet_req', null, 'nutr', 'rowlower');
insert into sysblocks values ('diet', 'diet_req_ub', 'diet_req', null, 'nutr', 'rowupper');
insert into sysblocks values ('diet', 'diet_buyreq_block', 'diet_req', 'diet_buy',
                              'amount', 'blockdata');

rem ############
rem COLUMN VIEWS
rem ############

create view diet_buy(ix, food) as
select rownum, food
from syn_diet_food;

create view diet_buy_obj(row_ix,col_ix,coef) as
select -3, ix, cost
from diet_buy, syn_diet_food
where diet_buy.food = syn_diet_food.food;
```

```
create view diet_buy_lb(row_ix,col_ix,coef) as
select -2, ix, f_min
from diet_buy, syn_diet_food
where diet_buy.food = syn_diet_food.food;

create view diet_buy_ub(row_ix,col_ix,coef) as
select -1, ix, f_max
from diet_buy, syn_diet_food
where diet_buy.food = syn_diet_food.food;

rem #########
rem ROW VIEWS
rem #########

create view diet_req(ix, vitamin) as
select rownum, vitamin
from syn_diet_nutr;

create view diet_req_lb(row_ix,col_ix,coef) as
select ix, -2, n_min
from diet_req, syn_diet_nutr
where diet_req.vitamin = syn_diet_nutr.vitamin;

create view diet_req_ub(row_ix,col_ix,coef) as
select ix, -1, n_max
from diet_req, syn_diet_nutr
where diet_req.vitamin = syn_diet_nutr.vitamin;

rem ##################
rem MATRIX BLOCK VIEWS
rem ##################

create view diet_buyreq_block(row_ix, col_ix, coef) as
Select diet_req.ix, diet_buy.ix, amount
from diet_req, diet_buy, syn_diet_amount
where diet_req.vitamin = syn_diet_amount.vitamin and
      diet_buy.food = syn_diet_amount.food;
```

There is only one column strip in the model, corresponding to the decision variables of the quantities of the different foods to buy. The only constraint that these choices of food must satisfy is to meet the daily nutritional requirements, so there is also only one row strip in the model. The data tables describing an instance of the model are a table describing the minimum and maximum amounts of each vitamin that must be consumed, a table descibing a minimum and maximum amount of food a person can buy, as well as its cost, and a table telling the amount of each vitamin in a specific food.

The Army Model

The Army model is a classic problem in military manpower planning. We use the version presented in Wagner [Wag75] and [GM92]. Soldiers can be enlisted for any number of periods up to a certain maximum. The decision to be made is how many soldiers to employ of each enlistment length in each year to meet a required troop strength for every year of a planning horizon in order to minimize troop costs, where the costs are subject to inflation.

```
create table base_army_demand_data(
year number,
demand number
);

create table base_army_enlist_data(
length number,
cost number
);

create table base_army_infl_data(
year number,
infl number);

create synonym syn_army_demand_data for base_army_demand_data;
create synonym syn_army_enlist_data for base_army_enlist_data;
create synonym syn_army_infl_data for base_army_infl_data;

insert into sysmodels values ('army', 'min');

insert into sysdata values ('army', 'base_army_demand_data', 'syn_army_demand_data', null);
insert into sysdata values ('army', 'base_army_enlist_data', 'syn_army_enlist_data', null);
insert into sysdata values ('army', 'base_army_infl_data', 'syn_army_infl_data', null);

insert into sysrows values ('army', 'army_demand');
insert into syscols values ('army', 'army_enlist', 'integer');

insert into sysblocks values ('army', 'army_enlist_obj', null, 'army_enlist', null, 'colobj');
insert into sysblocks values ('army', 'army_demand_lb', 'army_demand', null, null,
                              'rowlower');
insert into sysblocks values ('army', 'army_block11', 'army_demand', 'army_enlist', null,
                              'blockdata');


rem ############
rem COLUMN VIEWS
rem ############

create view army_enlist(ix, year, length) as
```

```
select rownum, year, length
from syn_army_infl_data, syn_army_enlist_data;

create view army_enlist_obj(row_ix, col_ix, coef) as
select -3, ix, infl*cost
from army_enlist, syn_army_enlist_data, syn_army_infl_data
where army_enlist.year = syn_army_infl_data.year
and army_enlist.length = syn_army_enlist_data.length;

rem #########
rem ROW VIEWS
rem #########

create view army_demand(ix, year) as
select rownum, year
from syn_army_demand_data;

create view army_demand_lb(row_ix, col_ix, coef) as
select ix, -2, demand
from army_demand, syn_army_demand_data
where army_demand.year = syn_army_demand_data.year;

rem ##################
rem MATRIX BLOCK VIEWS
rem ##################

create view army_block11(row_ix, col_ix, coef) as
select army_demand.ix, army_enlist.ix, 1
from army_demand, army_enlist
where army_enlist.year + army_enlist.length - 1 >= army_demand.year and
army_enlist.year <= army_demand.year;
```

There is only column strip and row strip for this model. There are three data tables used to create an instance of the model. The first simply hold the required troop strength in each year. The second holds the (uninflated) cost per year of hiring a soldier of each potential enlistment length. The final table holds the estimated inflation factor for each year in the planning horizon.

The Steel Model

This problem is the multi-period steel production model `steelT2.mod` described in [FGK93]. The problem is to determine how many tons of steel to produce, send to inventory, and sell to maximize profits over many periods, while meeting constraints on the availability of the rolling mill.

```
rem ###########
```

```
rem BASE TABLES
rem ###########

create table stl_production_def (
product char(20),
initinv char(20),
rate number,
pcost number,
hcost number);

create table stl_markets_def (
product char(20),
period date,
revenue number,
limit number);

create table stl_periods_def (
period date,
avail number);

rem ########
rem SYNONYMS
rem ########

create synonym stl_production for stl_production_def;
create synonym stl_markets for stl_markets_def;
create synonym stl_periods for stl_periods_def;

rem ####################
rem UPDATE SYSTEM TABLES
rem ####################

insert into sysmodels values('steel', 'max');

insert into sysdata values('steel', 'stl_production_def', 'stl_production', null);
insert into sysdata values('steel', 'stl_markets_def', 'stl_markets', null);
insert into sysdata values('steel', 'stl_periods_def' ,'stl_periods', null);

insert into sysrows values('steel', 'stl_Time');
insert into sysrows values('steel', 'stl_BalanceFirst');
insert into sysrows values('steel', 'stl_Balance');

insert into syscols values('steel', 'stl_Make', 'CONTINUOUS');
insert into syscols values('steel', 'stl_Inv', 'CONTINUOUS');
insert into syscols values('steel', 'stl_Sell', 'CONTINUOUS');

insert into sysblocks values('steel', 'stl_Sell_ub', null, 'stl_Sell', null, 'colupper');
insert into sysblocks values('steel', 'stl_Time_ub', 'stl_Time', null, null, 'rowupper');
insert into sysblocks values('steel', 'stl_BalanceFirst_lb', 'stl_BalanceFirst', null, null,
```

```
                             'rowlower');
insert into sysblocks values('steel', 'stl_BalanceFirst_ub', 'stl_BalanceFirst', null, null,
                             'rowupper');
insert into sysblocks values('steel', 'stl_Balance_ub', 'stl_Balance', null, null,
                             'rowupper');
insert into sysblocks values('steel', 'stl_Balance_lb', 'stl_Balance', null, null,
                             'rowlower');

insert into sysblocks values('steel', 'stl_Make_obj', null, 'stl_Make', null, 'colobj');
insert into sysblocks values('steel', 'stl_Inv_obj', null, 'stl_Inv', null, 'colobj');
insert into sysblocks values('steel', 'stl_Sell_obj', null, 'stl_Sell', null, 'colobj');

insert into sysblocks values('steel', 'stl_block11', 'stl_Time', 'stl_Make', null,
                             'blockdata');
insert into sysblocks values('steel', 'stl_block21', 'stl_BalanceFirst', 'stl_Make', null,
                             'blockdata');
insert into sysblocks values('steel', 'stl_block22', 'stl_BalanceFirst', 'stl_Inv', null,
                             'blockdata');
insert into sysblocks values('steel', 'stl_block23', 'stl_BalanceFirst', 'stl_Sell', null,
                             'blockdata');
insert into sysblocks values('steel', 'stl_block31', 'stl_Balance', 'stl_Make', null,
                             'blockdata');
insert into sysblocks values('steel', 'stl_block32a', 'stl_Balance', 'stl_Inv', null,
                             'blockdata');
insert into sysblocks values('steel', 'stl_block32b', 'stl_Balance', 'stl_Inv', null,
                             'blockdata');
insert into sysblocks values('steel', 'stl_block33', 'stl_Balance', 'stl_Sell', null,
                             'blockdata');

rem ############
rem COLUMN VIEWS
rem ############

create view stl_Make (ix, product, period) as
select RowNum, X.product, period
from stl_markets X, stl_production
where X.product = stl_production.product;

create view stl_Make_obj (row_ix, col_ix, coef) as
select null, ix, -pcost
from stl_Make, stl_production
where stl_Make.product = stl_production.product;

create view stl_Inv (ix, product, period) as
select RowNum, X.product, period
from stl_markets X, stl_production
where X.product = stl_production.product;

create view stl_Inv_obj (row_ix, col_ix, coef) as
```

```
select null, ix, -hcost
from stl_Inv, stl_production
where stl_Inv.product = stl_production.product;

create view stl_Sell (ix, product, period) as
select RowNum, product, period
from stl_markets;

create view stl_Sell_obj (row_ix, col_ix, coef) as
select null, ix, revenue
from stl_Sell, stl_markets
where stl_Sell.product = stl_markets.product and
stl_Sell.period = stl_markets.period;

create view stl_Sell_ub (row_ix, col_ix, coef) as
select null, ix, limit
from stl_Sell, stl_markets
where stl_Sell.product = stl_markets.product and
stl_Sell.period = stl_markets.period;

rem #########
rem ROW VIEWS
rem #########

create view stl_Time (ix, period) as
select RowNum, stl_periods.period
from stl_periods;

create view stl_Time_ub (row_ix, col_ix, coef) as
select ix, null, avail
from stl_Time, stl_periods
where stl_time.period = stl_periods.period;

create view stl_BalanceFirst (ix, product) as
select RowNum, product
from stl_production;

create view stl_BalanceFirst_lb (row_ix, col_ix, coef) as
select ix, null, -initinv
from stl_BalanceFirst, stl_production
where stl_BalanceFirst.product = stl_production.product;

create view stl_BalanceFirst_ub (row_ix, col_ix, coef) as
select ix, null, -initinv
from stl_BalanceFirst, stl_production
where stl_BalanceFirst.product = stl_production.product;

create view stl_Balance (ix, product, period) as
select RowNum, product, period
```

```
from stl_markets
where period > (select min(period) from stl_markets);

create view stl_Balance_ub (row_ix, col_ix, coef) as
select ix, null, 0
from stl_Balance, stl_markets
where stl_balance.period = stl_markets.period and
stl_Balance.product = stl_markets.product;

create view stl_Balance_lb (row_ix, col_ix, coef) as
select ix, null, 0
from stl_Balance, stl_markets
where stl_balance.period = stl_markets.period and
stl_Balance.product = stl_markets.product;

rem ##################
rem MATRIX BLOCK VIEWS
rem ##################

create view stl_block11 (row_ix, col_ix, coef) as
select stl_Time.ix, stl_Make.ix, 1/rate
from stl_Time, stl_Make, stl_production
where stl_Time.period = stl_Make.period and
stl_Make.product = stl_production.product;

create view stl_block21 (row_ix, col_ix, coef) as
select stl_BalanceFirst.ix, stl_Make.ix, 1
from stl_BalanceFirst, stl_Make
where stl_BalanceFirst.product = stl_Make.product
and stl_Make.period = (select min(period) from stl_Make);

create view stl_block22 (row_ix, col_ix, coef) as
select stl_BalanceFirst.ix, stl_Inv.ix, -1
from stl_BalanceFirst, stl_Inv
where stl_BalanceFirst.product = stl_Inv.product
and stl_Inv.period = (select min(period) from stl_Inv);

create view stl_block23 (row_ix, col_ix, coef) as
select stl_BalanceFirst.ix, stl_Sell.ix, -1
from stl_BalanceFirst, stl_Sell
where stl_BalanceFirst.product =  stl_Sell.product
and stl_Sell.period = (select min(period) from stl_Sell);

create view stl_block31 (row_ix, col_ix, coef) as
select stl_Balance.ix, stl_Make.ix, 1
from stl_Balance, stl_Make
where stl_Balance.product = stl_Make.product and
stl_Balance.period = stl_Make.period;
```

```
create view stl_block32a (row_ix, col_ix, coef) as
select stl_Balance.ix, stl_Inv.ix, -1
from stl_Balance, stl_Inv
where stl_Balance.product = stl_Inv.product and
stl_Balance.period = stl_Inv.period;

create view stl_block32b (row_ix, col_ix, coef) as
select X.ix, stl_Inv.ix, 1
from stl_Balance X, stl_Inv
where X.product = stl_Inv.product and
stl_Inv.period = (select max(period) from stl_Inv where period < X.period);

create view stl_block33 (row_ix, col_ix, coef) as
select stl_Balance.ix, stl_Sell.ix, -1
from stl_Balance, stl_Sell
where stl_Balance.product = stl_Sell.product and
stl_Balance.period = stl_Sell.period;
```

There are three columns strips, corresponding to the distinct decisions of how much steel to make, send to inventory, and sell in each time period. There are three rowstrips. One corresponds to the mill availability constraints for each time period and the remaning two are necessary to balance the flow of products from time period to time period.

There are three tables holding data for an instance of the model. The `production` table holds for each product the initial inventory, the rate at which it can be produced, the production cost, and the holding cost. The `markets` table contains the revenue received and maximum amount that can be sold in each for each product and period. The `periods` table tells how much rolling mill time is available in each period.

This example gives an idea of how to deal with ordered sets of variables within **ARMOS** since we must balance the flow of steel from period to period within the model. Here we make use of the special SQL date functions to select the minimum or maximum period.

One other interesting feature of this model is this use of overlaid matrix blocks. Their use is often necessary whenever there is more than one set of technological coefficients for a block. Balance constraints, having both a +1 and -1 entry in the block, fall into this category.

Appendix B

In this appendix, we give a brief description of the commands and functionality of the ARMOS system.

`list`: This command displays the names of models stored in `SysModels` table.

`load <modelname>`: Loading a model is the first thing a user needs to do to access a model stored in system tables. If the model specified is not in the ARMOS database, an error message appears indicating this.

`matrix`: Once a model is successfully loaded, the block schematic view can be displayed by typing `matrix` command. This commands prints the column strip, row strip and block names of the model, in a schematic format. Below is the block schematic view of the production distribution model described earlier.

```
RDDOMS > matrix
            rowlower    produce     ship        assign      rowupper
prodrow     rlower1     block11     block12       0         rupper1
shiprow     rlower2       0         block22     block23     rupper2
centrow     rlower3       0           0         block33     rupper3
colupper                 cupper1     +inf        +inf
collower                   0           0           0
objective                obj1        obj2        obj3
```

`set <instancename>`: The system's instance management is performed by the `set` command. This command associates an instance name with the data tables used to create an instance of the problem. When this command is called, the system checks the `SysInstances` table to see whether there exists data table names specified for the instance name before. If the result is affirmative, those data table names are copied to the system table `SysData`. Otherwise, the user is asked to enter the data table names for this new instance of the model. These names are recorded to `SysInstances` and then copied to `SysData` immediately. After a model is loaded, the `set` command can be called many times to create different instances.

`show <blockname(s)>`: Once an instance is set, it is possible to view instance matrix for a particular block of the model with `show <blockname(s)>` command. This capability may be quite useful for analyzing the instance. Arguments to `show` can be a single block name or a list of block names. A dash as an argument stands for all the model blocks.

`solve`: This command is the heart of our system. It actually performs all the interaction with the solver. When the `solve` command is called, the system generates the actual matrix (triplets) for the instance and loads it to the solver. After the solution is found, it creates proper tables and puts back solution values associated with the column strips and row strips into these solution tables. The solution table names for the particular instance solved are recorded in `SysSols` for future reference. Several statistical information such as run date, run time, elapsed time are recorded in `SysRuns`.

`display <r/c> <tablename>`: To display the solution values obtained by the solver, the `display` command is used. The syntax of the display command is `display <r/c> <tablename>`, where `<tablename>` is the name of the particular solution table you wish to view, and `<r/c>` is "r" if the solution to be viewed is a row strip and "c" if it is a column strip.

`text <viewname>`: This command is for displaying the SQL commands used to create views for row and column strips and model blocks. This capability is particularly useful for report generation and debugging purposes at model generation phase. Argument to `text` can be a single view name or a list of view names. A dash as an argument stands for all the views used in model.